View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Illinois Digital Environment for Access to Learning and Scholarship Repository

Formal Verification of MSSP

Pierre Salverda
salverda@uiuc.edu

Craig Zilles
zilles@cs.uiuc.edu

December 16, 2003

Abstract

MSSP is a new execution paradigm that achieves high performance by removing correctness constraints from the critical path. A collection of concurrently executing slave processors, which are not on the critical path, check the operation of a single master processor, whose execution is on the critical path, but is fast because it need not be correct. This report formally verifies that such an execution model works, in the sense that it correctly achieves a sequential execution of the application code.

We describe abstract models of both the sequential and MSSP execution paradigms, and distill from these the fundamental aspects of functionality that are needed to establish their equivalence. The verification itself is an iterative process. We begin with a number of high-level assumptions, which we use to prove some very basic results, and then successively refine our formalisms to argue that our initial assumptions are indeed reasonable.

In formally reasoning about MSSP, we achieve a number of goals. First, and most importantly, we demonstrate that MSSP is indeed equivalent to sequential execution. Second, we derive an abstract model of MSSP execution, which permits us to distill the fundamental properties of an MSSP machine that are necessary for correct operation. Having thus enumerated those properties, we facilitate reasoning about the impacts on correctness (and hence the feasibility) of new design ideas. Finally, we show that operation of the master—which is not guaranteed to be correct—does not compromise overall correctness. We thus demonstrate that an architecture in which performance and correctness are pursued as distinct design goals is indeed viable, at least from the point of view of maintaining correctness irrespective of the activity of the performance sub-component.

1 Introduction

Master/Slave Speculative Parallelization (MSSP) [4] is a recent proposal for speculative parallelization of sequential programs. The paradigm uses a single processor, called the master, to spawn parallel tasks on multiple slave processors. Dataflow dependences between the tasks are resolved by the master, which predicts live-in values for each task by executing an *approximate* version of the original program. Slaves use the original program code when executing their appointed task, but consume the speculative live-ins supplied by the master. The results thus computed are permitted to affect the machine's architected (visible) state only when the live-ins used to compute them are consistent with the current architected state. If inconsistencies are detected, the results are discarded and the machine resumes its operation using the current, pristine architected state as a starting point.

Two factors faciliate high performance in MSSP. First, slave execution is truly concurrent because live-in values supplied by the master circumvent the inter-task dependences that would otherwise force serial execution. Second, the master's own execution can be made fast because it need only execute an approximate version of the original program. With sufficiently many slave processors, it is generally the master that determines overall MSSP performance. In turn, the master's contribution to performance is subject to two influences: the time spent executing the approximate code and the accuracy with which the approximate code models the original program. With aggressive optimization and appropriate selection of task boundaries, a highly efficient and very accurate approximate program can be obtained; overall, MSSP is able to achieve significant speed-ups over speculative, out-of-order superscalar machines [4].

A distinguishing feature of MSSP is its *decoupling* of performance and correctness concerns. That is, MSSP physically separates correctness and performance by devoting distinct hardware components to each. Figure 1 depicts this idea. Correctness is maintained by the slaves, which essentially check the results produced by the master. High performance is achieved by the master, which executes an aggressively optimized, but not necessarily correct, version of the program.

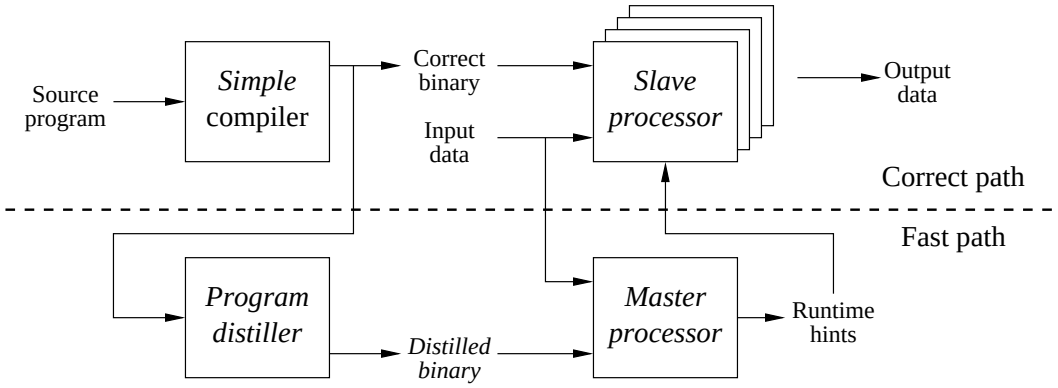

Figure 1: **Conceptual organization of an MSSP machine.** *The master executes an approximate program (the distilled binary) to run ahead of the slaves, providing hints (the live-ins) of where execution is likely to be headed. Slaves use those hints to concurrently compute the output results. To ensure correctness, live-ins are checked before the slaves' results are permanently committed to machine state.*

By separating correctness and performance, MSSP aims to address a problem faced by current architectures: performance and correctness tend to be opposing design goals; ensuring correctness demands simplicity, yet achieving performance demands complexity in the system. As noted above, concurrency in the slaves moves their performance off the critical path. Slaves can thus be engineered to be slow but simple, which makes their verification easier. In contrast, the master resides on the critical path, but, being unconstrained by correctness requirements, is amenable to complex optimizations (in both hardware and in the distilled binary) that aggressively target performance.

The primary goal of this work is to establish correctness of the extant MSSP proposal (as it stands in [3]). To this end, we demonstrate MSSP's equivalence to the existing sequential model, showing that any state reachable by an MSSP machine is also reachable by a sequential machine. We tackle the problem iteratively, developing first a high-level abstract model in which we make a number of simplifying assumptions. Section 3 describes this work. We then successively refine the formal models in Sections 4 and 5 to prove that our assumptions in the first step are indeed reasonable.

In developing an abstract model for MSSP operation, we achieve a secondary goal of distilling from the existing model—which is replete with technology-driven design trade-offs—a more basic model for MSSP that is devoid of implementation details. We intend to use this abstract model in reasoning about the correctness of subsequent iterations of MSSP design.

This work also serves an important role in terms of demonstrating the viability of an architecture that aims to decouple performance and correctness. More specifically, a machine successfully decouples performance and correctness if it meets two criteria. First, correct-path execution must not in any way be affected by the fast-path; that is, correctness must not be compromised by the pursuit of performance. Second, and of equal importance, performance on the fast path must be (largely) immune to the speed of the correct-path; that is, performance should not be constrained by correctness. It is the first of these two criteria that we demonstrate in this report: we prove that correctness in MSSP cannot be influenced by how the master operates, nor by the instructions contained in the distilled binary it executes.

Before we introduce the formal models for MSSP and sequential execution (Section 3), we describe the operation of MSSP in more detail in Section 2, which follows.

2 An overview of MSSP

In this section, we present an overview of Master/Slave Speculative Parallelization (MSSP). This high-level description is meant to provide the contextual knowledge necessary to understand the formal work that follows in the remainder of the paper. A more extensive treatment of MSSP can be found in [3] and [4].

Consider again Figure 1. An MSSP machine has two execution paths: the fast and the correct path. The fast path is composed of a single, complex processor—the *master*—that executes an aggressively optimized executable called the *distilled program*. The master processor runs ahead of the correct path execution to produce hints of

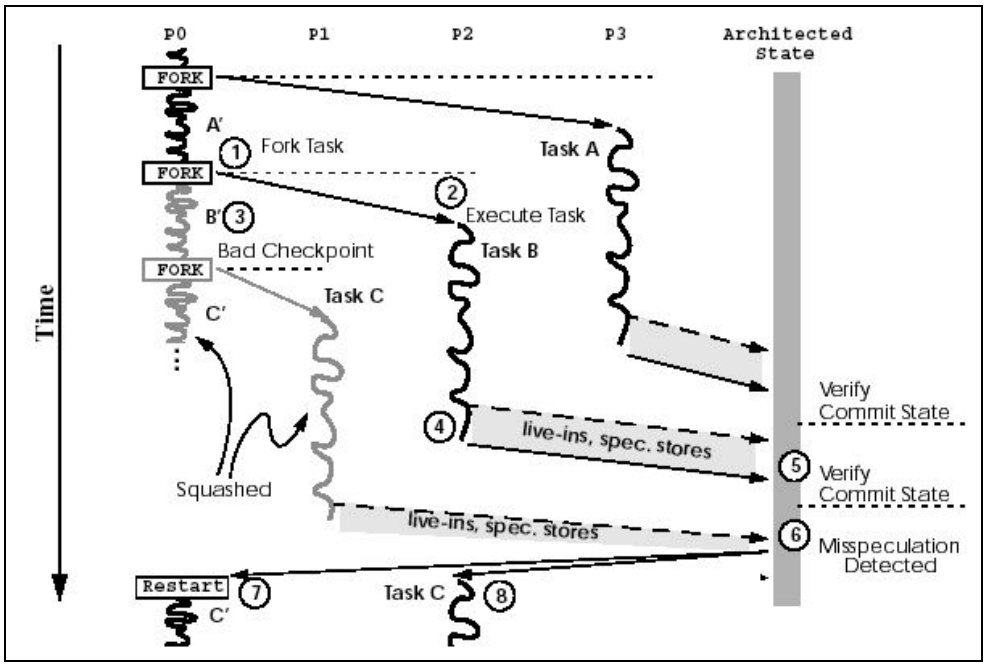

Figure 2: **Master processor distributes checkpoints to slaves.** *The master—executing the distilled program on processor P0—assigns tasks to slave processors, providing them with predicted live-in values in the form of checkpoints. The live-in values are verified when the previous task retires. Misspeculation, due to an incorrect checkpoint, causes the master to be restarted with the correct architected state.*

where the execution is headed. The correct path is implemented by multiple *slave* processors, which lag behind the master. Because the individual slave processors are slower than the master, we need a means for the correct path to keep up. MSSP uses *speculative parallelization* [2] for this purpose. Execution of the correct path program is split into segments, called *tasks*, that are executed concurrently on the slaves. From a correctness standpoint, the selection of task boundaries is relatively arbitrary, but it can affect performance.

To enable these tasks to execute independently and in parallel, the master execution is used to predict the sequence of tasks (*i.e.* the starting program counter (PC) of each task) and the live-in values to each task. These are the hints that the fast-path execution provides. The predictions are generated by logically taking a checkpoint of the master's state at the point corresponding to the beginning of the task. Low overhead checkpointing can be implemented by timestamping and buffering recent register writes and stores performed by the master, which serve as a "diff" from architected state; the master never updates architected state directly.

Because the master's predictions are not guaranteed to be correct, the slave processors do not update architected state immediately. Instead, each task's inputs (live-ins) and outputs (live-outs) are recorded and sent to the *verification/commit unit*. When a completed task becomes the oldest (that is, the next to commit), a memoization-like operation is performed that commits the outputs (live-outs) if the inputs match architected state.

Furthermore, because the master's predictions are not required, the slave processors can execute (in sequential mode) without the master. This allows the distilled program to be constructed for only a (performance critical) subset of the program. In effect, MSSP can be thought of as a mode that the system can shift into when a distilled program fragment exists for the current region of execution. Because MSSP mode is only a performance enhancement, neither the master nor the distiller need to handle the full functionality of the ISA.

2.1 MSSP example

To facilitate a conceptual understanding of MSSP, we provide an example that outlines its basic behavior. Figure 2 illustrates an MSSP execution with four processors: one master (P0) and three slaves (P1, P2, and P3) that begin the example idle. The master executes the distilled program and, at task boundaries, spawns a new task in the original program on an idle slave processor, providing it access to the buffered checkpoint values. At annotation (1) in the figure, the master processor spawns Task B onto processor P2. P2 begins executing the task after some

latency due to the inter-processor communication (2). P0 continues executing (3) the distilled program segment that corresponds to Task B, which we refer to as Task B'.

As the slave Task B executes, it will read values that it did not write (live-in values) and perform writes of its own (live-out values). If a corresponding checkpoint value is available, it is used for the live-in value; otherwise the value is read from current visible (architected) state. As the slave tasks are speculative—they are using predicted live-in values—their live-out values cannot be immediately committed: we have to ensure the live-in values are correct before committing the live-outs. To this end, we record the task's live-in and live-out values. When all previous tasks have completed and updated the architected state, the live-ins can be compared with the architected state. To avoid inter-processor communication in the verification critical path, our implementation performs this comparison at the (banked) shared level of the cache hierarchy. Thus, when the task is complete (4), P2 sends its live-in and live-out values to the shared cache. If the recorded live-in values exactly correspond to the architected state, then the task has been verified and can be committed, and the architected state can be updated (5) with the task's live-out values.

If one of the recorded live-in values differs from the corresponding value in the architected state (*e.g.* because the master wrote an incorrect value (3)), this mismatch will be detected during verification. On detection of the misspeculation (6), the master is squashed, as are all other in-flight tasks. At this time, the master is restarted at C' (7) using the current architected state. In parallel, non-speculative execution of the corresponding task in the original program (Task C) begins (8).

2.2 Program distillation

In both the master processor and the distilled program, we can exploit the fast path's lack of correctness constraints to implement optimizations that are not guaranteed to be correct. MSSP's fast-path compiler (the program distiller) exploits this property to eliminate predictable computations from the fast-path. The predictable program behaviors are removed via *approximation transformations*. Unlike traditional compiler transformations, which can only be applied in situations where they preserve all potential program behaviors, approximation transformations are allowed to arbitrarily violate correctness. By using profile information, we can apply these transformations such that the common case performance is improved and correctness violations are minimized.

To make the fast-path's hints useful to the correct path, the two binaries have to correspond at task boundaries. For example, if register `r11` is live-in to a task (*i.e.* the slave reads `r11` without defining it), the distilled program must include the necessary computation *before* the task boundary and store its result in `r11`. This correspondence requirement can potentially constrain program distillation, but the correspondence does not have to be exact; transition code—much like the "fix-up code" used by speculative compiler transformations, except that it need not be verified—can be used to transform the fast-path's state into a form expected by the correct path.

3 Proof of equivalence

In this section, we introduce and then prove the equivalence of abstract models for sequential and MSSP execution. We begin by formally defining a simplified model of sequential program execution, in which the operating system and I/O are absent. In Section 3.2, we then introduce a hypothetical machine that implements the MSSP execution model. We use this as the basis for our formal model of MSSP execution, which is presented in Section 3.3. By making a number of "reasonable assumptions", we then prove in Section 3.4 that the two models are equivalent. Sections 4 and 5 revisit a number of the simplifying assumptions we make here.

3.1 The SEQ execution model

The sequential execution model, SEQ, serves as a reference against which correctness of MSSP is measured. Since we do not wish to couple ourselves to a particular sequential ISA, we avoid specifying one for SEQ. That is, we define the operation of SEQ at a high level only. We can nonetheless reason about equivalence to MSSP because we can (reasonably) assume that the slaves implement the same ISA as our sequential reference machine; in other words, we show equivalence by assuming the same ISA in both models, but without interpreting that ISA.

Our formal models for both SEQ and MSSP are centred on the notion of machine state. The following subsection introduces this concept.

3.1.1 Machine state

Intuitively, a machine's state is captured by the collection of storage cells from which it is built. The state we are interested in is that which is visible via the machine's ISA. More specifically, this is the intersection of all storage cells that are:

- readable, directly or indirectly, by an instruction being executed, and hence can affect the outcome of that execution; or
- modifiable, directly or indirectly, through the execution of an instruction.

This characterization is of course not a precise one. We enumerate the above properties only to make explicit the intuition behind our formal notion of machine state: an *uninterpreted domain* over which sequential execution is defined. The precise definition follows.

Definition 3.1 ($\mathcal{S}_{seq}$) *We denote by $\mathcal{S}_{seq}$ the set of all states for sequential machines. At this point, the content of $\mathcal{S}_{seq}$ is uninterpreted.* □

Any configuration of a complete sequential machine is captured by some member of the domain $\mathcal{S}_{seq}$. However, the converse does not apply: a given $S \in \mathcal{S}_{seq}$ does not necessarily model the state of a complete machine; there exist machine states in $\mathcal{S}_{seq}$ that capture only a subset of a machine's complete state.

3.1.2 SEQ execution

Execution of an instruction results in updates to one or more storage cells. Accordingly, instruction execution constitutes a transformation of machine state. Sequential execution of more than one instruction is then naturally defined in terms of function composition.

Definition 3.2 (Sequential execution) *State transition function $seq : \mathcal{S}_{seq} \times \mathbb{Z}^+ \mapsto \mathcal{S}_{seq}$ models the sequential execution of one or more instructions. We define seq inductively for all $n \geq 0$.*

$$\begin{aligned} seq(S, n+1) &= \left\{ \begin{array}{ll} seq\_step(seq(S,n)) & \text{if } n \geq 1 \\ seq\_step(S) & \text{otherwise} \end{array} \right] \\ seq(S, 0) &= S \end{aligned}$$

The function $seq\_step : \mathcal{S}_{seq} \mapsto \mathcal{S}_{seq}$, which models the execution of a single instrcution, is uninterpreted. □

For S_0 and $S_1 \in \mathcal{S}_{seq}$, we will write $S_0 \overset{seq}{\rightsquigarrow} S_1$ if S_0 can be transformed into S_1 through the execution of some number of instructions. That is, $S_0 \overset{seq}{\rightsquigarrow} S_1$ if there exists n such that $seq(S_0, n) = S_1$. When n is known, we write $S_0 \overset{seq}{\rightsquigarrow}_n S_1$. Note that this definition states only that there exists some *number* of instructions that effect the transition, but it does not qualify *which* instructions will be involved. The initial machine state—in this case, S_0—determines those instructions implicitly. This stands to reason, since a machine's storage cells hold both instructions and data. Specifically, the program counter, itself a member of S_0, identifies the storage cell in which the encoding of the next instruction is held.

We now present a simple lemma that follows directly from the above definition of sequential execution.

Lemma 3.1 *Let $n \geq 0$ and $S \in \mathcal{S}_{seq}$ be given. Then, for all $k \geq 0$, $seq(S, n+k) = seq(seq(S,n), k)$.* □

Proof By induction on k.

When $k = 0$, the result follows immediately from Definition 3.2: for any $n \geq 0$ and $S \in \mathcal{S}_{seq}$, $seq(S, n+0) = seq(S,n) = seq(seq(S,n), 0)$.

For the induction step, we assume that $seq(S, n+k) = seq(seq(S,n), k)$ for some $k \geq 0$. Consider then $seq(S, n+(k+1)) = seq(S, (n+k)+1)$. Applying Definition 3.2, this can be rewritten as $seq\_step(seq(S, n+k))$, which, by our assumption, is equal to $seq\_step(seq(seq(S,n), k))$. Applying again Definition 3.2, this time in reverse, we get $seq\_step(seq(seq(S,n), k)) = seq(seq(S,n), k+1)$, as required. ■

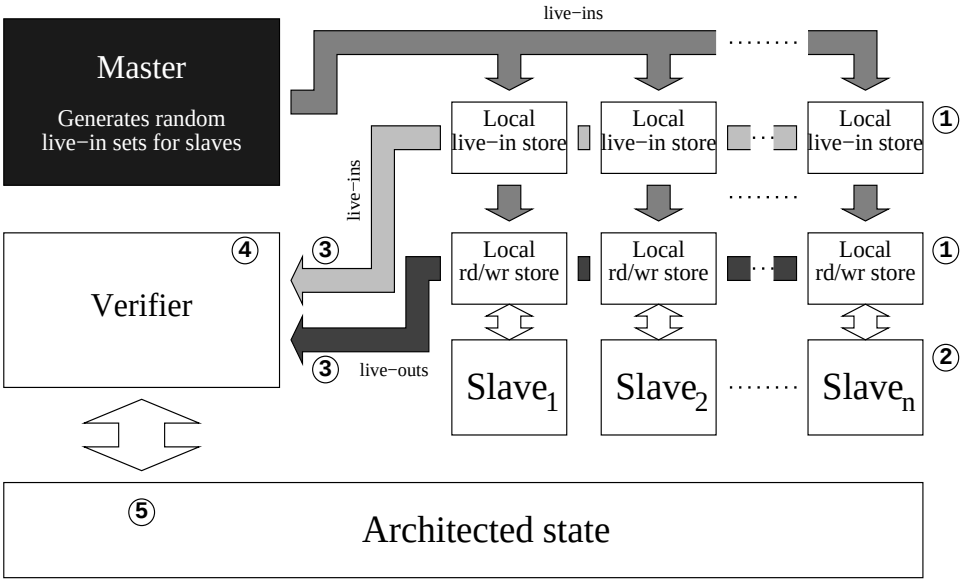

Figure 3: **Conceptual MSSP machine used for formal verification.**

3.2 An abstract MSSP machine

We now introduce an abstract version of an MSSP machine that will serve as the foundation upon which we build our formal model of MSSP execution. We stress that, while this abstract machine captures the salient features of MSSP operation, it is not intended to serve as a blueprint for a real implementation. Specifically, we pay no heed to performance concerns in our model; we are interested here in correctness, and accordingly organize our design so as to facilitate abtract reasoning about MSSP. Figure 3 depicts the machine.

Clearly, this differs in a number of respects from the MSSP infrastructure described in Section 2. Most notably, the master processor is viewed now as a "black box" that generates live-in sets for the slaves. As such, the formal model makes no assumptions about the live-in data used by slaves; the master is modelled essentially as a generator of *random* data.

Another divergence from the existing MSSP infrastructure is the manner in which slaves are associated with two local stores: one local live-in buffer and one local read/write store. The former, which cannot be read (directly) by a slave, holds the live-ins produced by the master for the task currently being executed. The latter constitutes the slave's local view of machine state. At the commencement of task execution, the content of the local live-in store is copied into the local read/write store (thus initializing it); all storage cells that can be read by the slave, or that have been written to by it, are contained in that store. We choose to model slaves in this manner because it permits us to make explicit two key properties of MSSP. First, a read-only record of *all* live-in data used by a slave is maintained at the slave. Second, writes generated during execution of a task are buffered locally—and coalesced, in cases of more than one write to the same storage cell—without being visible to any other slave and without affecting architected state.

At the completion of a task, a slave sends the content of its two stores to the verify unit. The verifier checks the contents of the read-only store (the live-ins) against architected state. If verification succeeds, the contents of the writable store (the results of slave execution) are committed to architected state; if not, they are simply discarded.

The preceding description of MSSP operation uncovers a set of requirements that must hold if correct operation of our abstract MSSP machine is to be ensured. Those requirements, which are annotated in Figure 3, are as follows.

① **Local stores:** A local live-in store may hold only what the master supplied at the commencement of a task, and this content may not change for the duration of that task. Similarly, the local read/write store is intialized with the content of the live-in store, and, thereafter, may be updated only with data that is produced by the slave during execution of its task.

② **Slave processors:** All slaves must be correct implementations of a given sequential ISA. Slaves may read from

and write to their local read/write store only.

③ **Interconnect:** Live-ins and live-outs must be transferred, without modification, to the verify/commit unit at the end of a task. That is, the data received by the verifier must be exactly the data held in the originating store.

④ **Verify/Commit:** The verifier must be a correct implementation of the verification functionality (we formalize this behavior shortly). Intuitively, this involves permitting updates to architected state only when *all* live-ins are consistent with the current architected state.

⑤ **Architected state:** Naturally, architected state must correctly hold all data written to it by the commit unit. Further, *only* the commit unit is able to write to architected state.

We will henceforth assume all of the above hold.

3.3 The MSSP execution model

Analogous to the sequential model, we define MSSP execution in terms of a state transition function, but we now make it explicit that it is the machine's architected state being manipulated. Further, manipulation of state occurs now at the granularity of tasks rather than instructions. We therefore begin by formalizing the notion of MSSP tasks.

3.3.1 Tasks

A task represents a unit of work in the MSSP model. Accordingly, we define below a formal construct to capture the idea that a task comprises a set of inputs (the live-ins produced by the master) and a set of outputs (which accrue as the slave executes).

Definition 3.3 (Task) *A task is a 5-tuple contained in* $\mathcal{T} = \mathcal{S}_{seq} \times \mathbb{Z}^+ \times \mathcal{S}_{seq} \times \mathbb{Z}^+ \times \mathcal{Q}$. $\langle S_{in}, n, S_{out}, k, q\rangle \in \mathcal{T}$ *denotes a task with live-in set* S_{in} *and live-out set* S_{out}. *The value* n, *which is fixed by the master, is the number of sequential instructions that constitute complete execution of this task;* k *is the number of instructions that have been executed by a slave so far (thus,* $0 \leq k \leq n$*). The fifth component,* q, *represents the current execution state of the task; we define* $\mathcal{Q} = \{\texttt{RUN}, \texttt{COMMIT}, \texttt{ABORT}\}$. □

Recall, at the creation of a new task, the master places live-in data in a slave's local live-in buffer, and this is copied to the local read/write store when the slave commences execution. Thus, a newly created task has form $\langle S_{in}, n, S_{in}, 0, \texttt{RUN}\rangle$, and, at its completion, has form $\langle S_{in}, n, S_{out}, n, \texttt{COMMIT}\rangle$; we will relate S_{in} and S_{out} later in this section. The use of state `ABORT` will be introduced in Section 4.

We define a number of functions on $\mathcal{T}$ for the sake of notational convenience. Let $T = \langle S_{in}, n, S_{out}, k, q\rangle$. Functions $\mathit{live\_in} : \mathcal{T} \mapsto \mathcal{S}_{seq}$ and $\mathit{live\_out} : \mathcal{T} \mapsto \mathcal{S}_{seq}$ produce the live-in and live-out sets for a given task. Thus, $\mathit{live\_in}(T) = S_{in}$ and $\mathit{live\_out}(T) = S_{out}$. Similarly, functions $\mathit{length} : \mathcal{T} \mapsto \mathbb{Z}^+$ and $\mathit{progress} : \mathcal{T} \mapsto \mathbb{Z}^+$ yield the second and fourth components of a task, respectively: $\mathit{length}(T) = n$ and $\mathit{progress}(T) = k$. The function $\mathit{state} : \mathcal{T} \mapsto \mathcal{Q}$ produces the execution state of a task: $\mathit{state}(T) = q$. We also define predicate $\mathit{done}(T)$ to be true if and only if $\mathit{state}(T) \neq \texttt{RUN}$.

Naturally, an MSSP machine operates on more than one task during the execution of a program. We thus introduce the notion of a sequence of tasks.

Definition 3.4 (Task sequence) *The set* $\mathcal{T}^*$ *denotes the set of all finite-length sequences of MSSP tasks. For* $T \in \mathcal{T}$, *we write* $[T]$ *to denote the singleton task sequence containing* T *only. The empty task sequence is denoted* ϵ. *Operator* $|$ *concatenates task sequences. Thus, for* $\tau \in \mathcal{T}^*$ *and* $T \in \mathcal{T}$, $[T]|\tau$ *is a new sequence formed by prepending* T *to* τ. *Also,* $\tau|\epsilon = \epsilon|\tau = \tau$. □

3.3.2 MSSP execution

Having defined tasks and task sequences, we are now ready to define formally the abstract machine's overall operation. We do so through a series of definitions.

Definition 3.5 (MSSP execution) *State transition function* $mssp : \mathcal{S}_{seq} \times \mathcal{T}^* \mapsto \mathcal{S}_{seq}$ *captures MSSP execution:*

$$mssp(A, [T]|\tau) = \left\{ \begin{array}{ll} mssp(commit(A,T),\tau) & \text{if } done(T) \\ mssp(A, mssp\_step([T]|\tau)) & \text{otherwise} \end{array} \right]$$

$$mssp(A, \epsilon) = A$$

Function $mssp\_step : \mathcal{T}^* \mapsto \mathcal{T}^*$ *captures the activity of the slaves and is defined as follows.*

$$mssp\_step([T_1, T_2, \ldots, T_m]) = [T_1, T_2, \ldots, slave\_step(T_i), \ldots, T_m]$$

for all $1 \leq i \leq m$ *for which* $done(T_i)$ *is false.* □

The transition function defines MSSP operation at a coarse granularity: the machine either commits completed tasks to architected state, or it simply advances the state of any tasks that are not yet completed. We will write $S_0 \overset{mssp}{\rightsquigarrow} S_1$ if there exists a $\tau \in \mathcal{T}^*$ such that $mssp(S_0, \tau) = S_1$; if τ is known, we write $S_0 \overset{mssp}{\rightsquigarrow}_\tau S_1$.

Recall, we stipulated in Section 3.2 that slaves execute according to the SEQ model. This requirement is captured formally in the following definition.

Definition 3.6 (Slave execution) *Execution of a slave processor is modelled by function* $slave\_step : \mathcal{T} \mapsto \mathcal{T}$. *For* $T \in \mathcal{T}$ *such that* $state(T) = \texttt{RUN}$, *we define:*

$$slave\_step(\langle S_{in}, n, S_{out}, k, \texttt{RUN}\rangle) = \left\{ \begin{array}{ll} \langle S_{in}, n, seq\_step(S_{out}), k+1, \texttt{RUN}\rangle & \text{if } k < n \\ \langle S_{in}, n, S_{out}, k, \texttt{COMMIT}\rangle & \text{otherwise} \end{array} \right]$$

The function is undefined for all tasks T *with* $state(T) \neq \texttt{RUN}$. □

Note that the above definition exploits the fact that our abstract machine initializes the read/write store with the live-in set at the commencement of a task. More precisely, because the slave's local read/write store begins with all state supplied by the master, the first step in slave execution simply advances that set using the sequential model: $live\_in(T)$ is transformed to $seq\_step(live\_in(T))$. Subsequent steps will likewise advance the set. Extrapolating, we arrive at the following lemma.

Lemma 3.2 *For any* $T \in \mathcal{T}$, $live\_out(T) = seq(live\_in(T), progress(T))$. □

Proof By induction on k, the details of which we omit. ■

Note that the above lemma applies throughout the lifetime of any given task. In particular, it applies at task completion, in which case $live\_out(T) = seq(live\_in(T), length(T))$.

The verify/commit unit updates architected state with the live-out set of a task when it is complete (subject to certain conditions being met, of course). We call this process *superimposition* of live-outs on architected state.

Definition 3.7 (Sumperimposition) *Operator* $\leftarrow : \mathcal{S}_{seq} \mapsto \mathcal{S}_{seq}$ *denotes the superimposition of machine state. It remains uninterpreted at this point.* □

That is, $S_0 \leftarrow S_1$ denotes the superimposition of machine state S_1 on S_0. We do not interpret this operation in this section simply because the domain $\mathcal{S}_{seq}$ itself remains uninterpreted. Nonetheless, it is useful to understand the intuition behind its working: $S_0 \leftarrow S_1$ is a new machine state in which values held by storage cells in S_1 "override" values currently held in those cells in S_0. It is permitted that S_0 refer to cells not currently held by S_1, in which case $S_0 \leftarrow S_1$ will contain that same state, unaffected by the superimposition. Likewise, if S_1 contains state not held by S_0, the superimposition operator effectively introduces that state into the resulting set $S_0 \leftarrow S_1$.

Having thus introduced and informally characterized superimposition, we can now define the conditions under which the verify unit applies it. Intuitively, we will permit a task to be committed to (superimposed on) architected state only if the resulting state is the same as that which would have been produced by sequential execution.

Definition 3.8 (Task safety) *We say* $T \in \mathcal{T}$ *is safe for* $A \in \mathcal{S}_{seq}$, *and write* $safe(A, T)$, *if* $seq(A, length(T)) = A \leftarrow live\_out(T)$. *Extending this idea to task sequences, we say* $[T]|\tau \in \mathcal{T}^*$ *is safe for* $A \in \mathcal{S}_{seq}$ *if* $safe(A, T)$ *and* τ *is safe for* $seq(A, length(T))$. □

Since $live\_out(T) = seq(live\_in(T), length(T))$ at the completion of T (Lemma 3.2), task safety equivalently characterized by the requirement $seq(A, length(T)) = A \leftarrow seq(live\_in(T), length(T))$.

Task safety underpins MSSP correctness, and we accordingly devote considerable attention to it in Section 4. For now we treat it as a basic property that can be checked by the verify/commit unit. Whence the following definition, which completes our formalization of MSSP execution.

Definition 3.9 (Commit) *The function* $commit : \mathcal{S}_{seq} \times \mathcal{T} \mapsto \mathcal{S}_{seq}$*, which denotes the action of the verify/commit unit, is defined:*

$$commit(A, T) = \left\{ \begin{array}{ll} A \leftarrow live\_out(T) & \text{if } safe(A, T) \\ A & \text{otherwise} \end{array} \right]$$

□

3.4 Equivalence

To prove that MSSP is equivalent to SEQ, we show that any transformation of machine state that can be effected by MSSP has a corresponding transformation in SEQ. That is, we show $S_0 \overset{mssp}{\rightsquigarrow} S_1 \Rightarrow S_0 \overset{seq}{\rightsquigarrow} S_1$.

3.4.1 Equivalence with safe sequences

We begin by proving a slightly weaker claim: we show that equivalence holds for a safe sequence of tasks. Extending to any sequence of tasks, which is an easy step, is covered in the next sub-section.

Theorem 3.1 *Let* $\tau = [T_1, T_2, \ldots, T_m]$ *be a non-empty sequence of tasks, and let* A_0 *be a machine state for which* τ *is safe. Then:*

$$mssp(A_0, \tau) = seq(A_0, \sum_{i=1}^{m} length(T_i))$$

□

Proof We will write A_k as shorthand for $seq(A_0, \sum_{i=1}^{k} length(T_i))$. Our proof obligation is thus to show that $mssp(A_0, [T_1, T_2, \ldots, T_k]) = A_k$. We use induction on k for this purpose.

The base case follows directly from our definition of task safety. Let A_0 be some machine state for which $\tau = [T_1]$ is safe. If the sequence $[T_1]$ is safe, it follows (from Definition 3.8) that the task T_1 is itself safe for A_0, and hence that $seq(A_0, length(T_1)) = A_0 \leftarrow seq(live\_in(T_1), length(T_1))$. That is, $A_1 = A_0 \leftarrow seq(live\_in(T_1), length(T_1))$. The right-hand side of this expression is easily rewritten (using Lemma 3.2) as $A_0 \leftarrow live\_out(T_1)$, which, by Definition 3.9, is equal to $commit(A_0, T_1)$. Definition 3.5 ensures $commit(A_0, T_1) = mssp(A_0, [T_1])$.

The inductive step proceeds similarly. Assume the result holds for some $k \geq 1$. That is, assume $A_k = mssp(A_0, \tau)$, where $\tau = [T_1, T_2, \ldots, T_k]$ is safe for A_0. Then consider task sequence $\tau' = \tau | [T_{k+1}]$ that is also safe for A_0. We want to show $mssp(A_0, \tau') = A_{k+1}$. Applying Definition 3.8 multiple times, safety of τ' implies T_{k+1} is safe for $seq(\ldots seq(seq(A_0, length(T_1)), length(T_2)), \ldots, length(T_k))$. Using Lemma 3.1, we can rewrite this expression as $seq(A_0, \sum_{i=1}^{k} length(T_i))$. That is, T_{k+1} is safe for A_k. This, in turn, implies $seq(A_k, length(T_{k+1})) = A_k \leftarrow live\_out(T)$. From our hypothesis, it follows that $seq(A_k, length(T_{k+1})) = mssp(A_0, \tau) \leftarrow live\_out(T_{k+1})$. The right-hand side of this expression is simply $mssp(A_0, \tau | [T_{k+1}])$; the left-hand side is, by definition, A_{k+1}. ■

3.4.2 Equivalence for all sequences

We can easily extend the above proof to cater for any sequence of tasks, thus concluding our proof that MSSP and SEQ are equivalent.

Theorem 3.2 *Let* A_0*,* $A_1 \in \mathcal{S}_{seq}$ *and* $\tau \in \mathcal{T}^*$*. Then*

$$A_0 \overset{mssp}{\rightsquigarrow}_{\tau} A_1 \Rightarrow \exists n \geq 0 : A_0 \overset{seq}{\rightsquigarrow}_{n} A_1$$

□

Proof Assume $A_0 \overset{mssp}{\rightsquigarrow}_\tau A_1$. It follows from our definition of MSSP execution—specifically, from Definition 3.9—that architected state is updated if and only if a task is safe for that state. There must therefore exist some subsequence of τ that effects the transition from A_0 to A_1, and that sequence contains only safe tasks. Let τ' be that subsequence. Since those members of τ that are not safe do not effect any state changes, it follows that $mssp(A_0, \tau) = mssp(A_0, \tau') = A_1$. We can then apply Theorem 3.1 to show that $mssp(A_0, \tau') = seq(A_0, \sum_{i=1}^{k} length(T_i))$, where $\tau' = [T_1, T_2, \ldots, T_k]$. That is, the value of n, whose existence is asserted by the theorem, is $\sum_{i=1}^{k} length(T_i)$. ■

4 Task safety

In the previous section, we proved that MSSP and SEQ are equivalent by making the (reasonable) assumption that task safety is a property that can be checked by the verify unit. In this section, we refine our formal models to prove that a more low-level set of checks—which are realistically assumed to be performed by the hardware—can be used to ensure task safety holds. To do so, we must enhance our formalisms to interpret the domain $\mathcal{S}_{seq}$ and hence define more rigorously the superimposition operator. This will, in turn, allow us to infer some key properties of sequential execution, which together can be used to reason about the collection of writes that accrue in a slave's read/write store. We begin with machine state.

4.1 Machine state

We have so far viewed machine state merely as a set that captures the values held in a machine's various storage cells. To formalize this intuition, we define the content of a machine state to be a collection of name-value pairs. The name component of each pair represents a unique identifier for one of the machine's storage cells; the value component represents the contents of that storage cell.

Definition 4.1 (Machine state) *The set of all machine states $\mathcal{S}_{seq}$ is defined*

$$\mathcal{S}_{seq} = \{S \subseteq \mathcal{N}_{seq} \times \mathcal{V}_{seq} : well\_formed(S)\}$$

The sets $\mathcal{N}_{seq}$ and $\mathcal{V}_{seq}$, which are uninterpreted, hold, respectively, the names of and values that can be held in a machine's storage cells. For $S \subseteq \mathcal{N}_{seq} \times \mathcal{V}_{seq}$, we define predicate well_formed(S) to be true if and only if $(n, v_1) \in S$ and $(n, v_2) \in S \Rightarrow v_1 = v_2$. □

Thus, a machine state is a set of uniquely named pairs, each of which defines the value held in one of the machine's storage cells. The exact set of names and allowable values are not specified simply because we do not want to define a particular ISA. We need understand only that the set $\mathcal{N}_{seq}$ constitutes the set of storage cells identified by the ISA, and, as such, comprises general-purpose registers, the program counter, and main memory addresses.

We are now in a position to define the superimposition operator formally. Before we do so, however, we introduce some terminology that will prove useful shortly.

Definition 4.2 (Names) *Function $names : \mathcal{S}_{seq} \mapsto \mathcal{N}_{seq}$ produces the names of all pairs contained in a given machine state. That is,*

$$names(S) = \{n \in \mathcal{N}_{seq} : \exists (n, v) \in S \text{ for some } v \in \mathcal{V}_{seq}\}$$

□

Definition 4.3 (Cover) *We say that $S \in \mathcal{S}_{seq}$ covers $N \in \mathcal{N}_{seq}$ if $N \subseteq names(S)$.* □

4.2 Superimposition

Recall, operator $\leftarrow\!\!:\ \mathcal{S}_{seq} \times \mathcal{S}_{seq} \mapsto \mathcal{S}_{seq}$ captures the action of the commit unit. We previously described its operation informally, stating that it replaces appropriate members in its left operand with the content of its right operand. Having formalized the notion of machine state, we can now interpret this operation more precisely.

Definition 4.4 (Superimposition) *For S_1, $S_2 \in \mathcal{S}_{seq}$, we define*

$$S_1 \leftarrow S_2 = (S_1 \ominus S_2) \cup S_2$$

where $S_1 \ominus S_2 = S_1 - \{(n, v) \in S_1 : n \in names(S_1) \cap names(S_2)\}$. □

Informally, the superimposition of S_2 onto S_1 is similar to the union of the two sets, but is *biased* in the sense that members of S_2 are always chosen over members of S_1, whenever there is a choice. This intuitive characterization of superimposition is expressed more formally in the following lemma, the proof of which follows directly from Definition 4.4.

Lemma 4.1 *If $(n, v) \in S_1 \leftarrow S_2$, then exactly one of the following is true.*

1. *$(n, v) \in S_1$ and S_2 does not cover $\{n\}$; or*
2. *$(n, v) \in S_2$.*

□

That is, (n, v) is in $S_1 \leftarrow S_2$ if S_1 already contains it and S_2 has no such member, or (n, v) is in S_2. Note that in the latter case, S_1 may or may not contain a pair with name n; it does not make any difference because the like-named pair in S_2 supercedes it.

Lemma 4.1 allows us to infer two important properties of operator $\leftarrow$, namely its associativity and its preservation of set containment. The following two lemmas present these results.

Lemma 4.2 *The superimposition operator is associative. That is, if S_1, S_2 and $S_3 \in \mathcal{S}_{seq}$, then*

$$(S_1 \leftarrow S_2) \leftarrow S_3 = S_1 \leftarrow (S_2 \leftarrow S_3)$$

□

Proof We show mutual containment.

$(S_1 \leftarrow S_2) \leftarrow S_3 \subseteq S_1 \leftarrow (S_2 \leftarrow S_3)$. Let $(n, v) \in (S_1 \leftarrow S_2) \leftarrow S_3$. Then, by Lemma 4.1, either $(n, v) \in (S_1 \leftarrow S_2)$ and S_3 does not cover $\{n\}$, or $(n, v) \in S_3$. We consider each case in turn.

- If $(n, v) \in (S_1 \leftarrow S_2)$ then, again by Lemma 4.1, one of the following must hold:
  - $(n, v) \in S_1$ and S_2 does not cover $\{n\}$; or
  - $(n, v) \in S_2$.

  In the first case, we know $(n, v) \in S_1$ and neither S_2 nor S_3 covers $\{n\}$. Hence, $S_2 \leftarrow S_3$ cannot cover $\{n\}$ either, and so $(n, v) \in S_1 \leftarrow (S_2 \leftarrow S_3)$. In the second case, S_3 does not cover $\{n\}$, but $(n, v) \in S_2$, and so $(n, v) \in (S_2 \leftarrow S_3)$; it follows that $(n, v) \in S_1 \leftarrow (S_2 \leftarrow S_3)$.
- Since $(n, v) \in S_3$, it follows directly from Definition 4.4 that $(n, v) \in (S_2 \leftarrow S_3)$. The latter implies, in turn, that $(n, v) \in S_1 \leftarrow (S_2 \leftarrow S_3)$.

In all cases, $(n, v) \in (S_1 \leftarrow S_2) \leftarrow S_3 \Rightarrow (n, v) \in S_1 \leftarrow (S_2 \leftarrow S_3)$. That is, $(S_1 \leftarrow S_2) \leftarrow S_3 \subseteq S_1 \leftarrow (S_2 \leftarrow S_3)$.

$S_1 \leftarrow (S_2 \leftarrow S_3) \subseteq (S_1 \leftarrow S_2) \leftarrow S_3$. The argument for containment in the other direction proceeds along the same lines, and so we will not repeat the details here.

Having established mutual containment, it follows that $S_1 \leftarrow (S_2 \leftarrow S_3) = (S_1 \leftarrow S_2) \leftarrow S_3$. ■

Lemma 4.3 *The superimposition operator preserves set containment. That is, for S_1, S_2 and $S_3 \in \mathcal{S}_{seq}$,*

$$S_1 \subseteq S_2 \Rightarrow (S_1 \leftarrow S_3) \subseteq (S_2 \leftarrow S_3)$$

□

Proof Assume $(n, v) \in S_1 \leftarrow S_3$. Then, by Lemma 4.1, either $(n, v) \in S_1$ and S_3 does not cover $\{n\}$, or $(n, v) \in S_3$. In the first case, containment of S_1 in S_2 ensures that (n, v) is in S_2 also; since S_3 does not cover $\{n\}$, it follows that $(n, v) \in S_2 \leftarrow S_3$ too. In the second case, $(n, v) \in S_3$ guarantees $(n, v) \in S_2 \leftarrow S_3$. ■

4.3 Sequential execution

Conceptually, a sequential machine operates by fetching an instruction from machine state, decoding and executing it, and then writing the results back to machine state. This view leads naturally to a definition of instruction execution in terms of superimposition.

Definition 4.5 (Instruction execution) *The execution of a single instruction in the sequential execution model is defined:*

$$seq\_step(S) = S \leftarrow \delta(S)$$

□

The set $\delta(S) \in \mathcal{S}_{seq}$ is a collection of name-value pairs that constitute the changes to state that will result from executing the next instruction. Effectively, the function $\delta : \mathcal{S}_{seq} \mapsto \mathcal{S}_{seq}$ performs the fetch-decode-execute steps alluded to above; the superimposition achieves the final write of the results back to machine state.

Although we do not interpret function δ at this point, we do stipulate that there are preconditions for its meaningful application. This is necessary because, in our MSSP model, slave processors operate on machine state that is potentially unsuitable for the purposes of executing the next instruction[1]. This leads us to the following definition.

Definition 4.6 (Completeness) *We say $S \in \mathcal{S}_{seq}$ is complete if $\delta(S)$ is defined. We inductively extend this definition to say S is n-complete if S is complete and $seq(S, 1)$ is $(n-1)$-complete.* □

Informally, S is complete if it contains all name-value pairs needed for computation of $\delta(S)$, and hence for the execution of the next instruction. If S is n-complete then execution of the next n instructions is well-defined.

Having defined execution of a single instruction in terms of superimposition, we extend the idea naturally to sequential execution of more than one instruction.

Definition 4.7 (Cumulative writes) *The function $\Delta : \mathcal{S}_{seq} \times \mathbb{Z}^+ \mapsto \mathcal{S}_{seq}$ produces the cumulative writes that accrue from the sequential execution of multiple instructions. It is defined inductively for all $n \geq 0$, as follows.*

$$\begin{aligned} \Delta(S, n+1) &= \left\{ \begin{array}{ll} \Delta(S,n) \leftarrow \delta(seq(S,n)) & \text{if } n >= 1 \\ \delta(S) & \text{otherwise} \end{array} \right] \\ \Delta(S, 0) &= \emptyset \end{aligned}$$

□

The set of cumulative writes is simply the collection of updates made to the slave's local read/write store. In cases where a given storage cell has been updated more than once, the cumulative writes define only its latest value (because writes are *coalesced* by the superimposition operator).

Definition 4.7 permits us to characterize sequential execution in terms of superimposition. The following lemma presents this important result.

Lemma 4.4 (Sequential execution through cumulative writes) *Let $S \in \mathcal{S}_{seq}$ be n-complete ($n \geq 0$). Then:*

$$seq(S, n) = S \leftarrow \Delta(S, n)$$

□

Proof By induction on n.

The base case is immediate: $seq(S, 0) = S = S \leftarrow \emptyset = S \leftarrow \Delta(S, 0)$.

For the inductive step, we assume $seq(S, k) = S \leftarrow \Delta(S, k)$ for some $k \geq 0$. Now, consider $seq(S, k+1)$. By definition, $seq(S, k+1) = seq\_step(seq(S, k))$. Applying Definition 4.5 to the right hand side of this expression, we get $seq(S, k) \leftarrow \delta(seq(S, k))$. From our hypothesis, this is the same as $(S \leftarrow \Delta(S, k)) \leftarrow \delta(seq(S, k))$. Applying Lemma 4.2, we can rewrite the latter as $S \leftarrow (\Delta(S, k) \leftarrow \delta(seq(S, k)))$. But $\Delta(S, k) \leftarrow \delta(seq(S, k))$ is, by Definition 4.7, equal to $\Delta(S, k+1)$. Thus, $seq(S, k+1) = S \leftarrow \Delta(S, k+1)$, and we are done. ■

[1] Recall, we make no assumptions about the about the live-ins produced by the master, and so it cannot be relied upon that all machine state needed for the execution of the next instruction is available to the slave in its local store.

An important result that follows from the above characterization of sequential execution is that two machine states that are both complete and consistent with one another will remain consistent with one another after the sequential execution of one or more instructions. By "consistent" we mean that the two sets agree on the values of storage cells whose names are common to both. The following lemma formalizes this.

Lemma 4.5 *Let S_1, $S_2 \in \mathcal{S}_{seq}$ be n-complete machine states (for some $n \geq 0$). Then:*

$$S_1 \subseteq S_2 \Rightarrow seq(S_1, n) \subseteq seq(S_2, n)$$

□

Proof The result follows directly from a simple, inductive extension of Lemma 4.3.

The base case is trivial: $S_1 = seq(S_1, 0)$ and similarly for S_2. Thus, $S_1 \subseteq S_2 \Rightarrow seq(S_1, 0) \subseteq seq(S_2, 0)$.

For the inductive step, assume $seq(S_1, k) \subseteq seq(S_2, k)$ for some $0 \leq k \leq n$. Consider then $seq(S_1, k+1) = seq\_step(seq(S_1, k)) = seq(S_1, k) \leftarrow \delta(seq(S_1, k))$ (Definitions 3.2 and 4.5, respectively). Likewise, $seq(S_2, k+1) = seq(S_2, k) \leftarrow \delta(seq(S_2, k))$. It follows from Definition 4.6 that, because both S_1 and S_2 are n-complete, $seq(S_1, k)$ and $seq(S_2, k)$ are also 1-complete. We have assumed $seq(S_1, k) \subseteq seq(S_2, k)$, so both sets must agree on the next instruction to be executed. Since instruction execution is deterministic, this, in turn, implies $\delta(seq(S_1, k)) = \delta(seq(S_2, k))$. We can then apply Lemma 4.3 to get $seq(S_1, k) \leftarrow \delta(seq(S_1, k)) \subseteq seq(S_2, k) \leftarrow \delta(seq(S_2, k))$, and hence that $seq(S_1, k+1) \subseteq seq(S_2, k+1)$. ■

We established in the above proof that $S_1 \subseteq S_2 \Rightarrow \delta(S_1) = \delta(S_2)$. This result can, in fact, be generalized to cumulative writes.

Lemma 4.6 *Let S_1, $S_2 \in \mathcal{S}_{seq}$ be n-complete machine states ($n \geq 0$). Then*

$$S_1 \subseteq S_2 \Rightarrow \Delta(S_1, n) = \Delta(S_2, n)$$

□

Proof A simple inductive extension of our arguments in the proof of Lemma 4.5.

Assume $\Delta(S_1, k) = \Delta(S_2, k)$, where $S_1 \subseteq S_2$ and $k \geq 0$. Lemma 4.5 guarantees that $seq(S_1, k) \subseteq seq(S_2, k)$. Then, using an argument that is by now familiar, this implies that $\delta(seq(S_1, k)) = \delta(seq(S_2, k))$. Combining this with our inductive hypothesis, we get $\Delta(S_1, k) \leftarrow \delta(seq(S_1, k)) = \Delta(S_2, k) \leftarrow \delta(seq(S_1, k))$. In other words, $\Delta(S_1, k+1) = \Delta(S_2, k+1)$. ■

4.4 Establishing task safety

In this sub-section, we show that task safety, the property we relied upon in Section 3, follows from two requirements:

1. *Completeness* — all machine state needed for well-defined execution of a task must be contained in that task's live-in set; and
2. *Containment* — live-in sets must be consistent with architected state.

This result is expressed formally in the following theorem.

Theorem 4.1 *Let $A, S \in \mathcal{S}_{seq}$. If S is n-complete and $S \subseteq A$, then $seq(A, n) = A \leftarrow seq(S, n)$.* □

Proof We know from Lemma 4.4 that, since S is n-complete, $seq(S, n) = S \leftarrow \Delta(S, n)$. Hence, $A \leftarrow seq(S, n) = A \leftarrow (S \leftarrow \Delta(S, n))$. By Lemma 4.2, the right hand side is the same as $(A \leftarrow S) \leftarrow \Delta(S, n)$. But $S \subseteq A$, so $A \leftarrow S = A$ (this can easily be inferred from Definition 4.4). Thus, $A \leftarrow seq(S, n) = A \leftarrow \Delta(S, n)$. Using Lemma 4.6 and the fact that $S \subseteq A$, we also know that $\Delta(A, n) = \Delta(S, n)$, and hence that $A \leftarrow seq(S, n) = A \leftarrow \Delta(A, n)$. Using Lemma 4.4 again, the latter expression is equivalent to $seq(A, n)$. ■

Corollary 4.7 *Let $A \in \mathcal{S}_{seq}$ denote an MSSP machine's architected state. If $T \in \mathcal{T}$ is a task such that live_in(T) is length(T)-complete and live_in(T) $\subseteq A$, then safe(A, T) is true.* □

Proof The preconditions for the corollary match those of Theorem 4.1, so it follows that $seq(A, length(T)) = A \leftarrow seq(live\_in(T), length(T))$. This is precisely the requirement for $safe(A, T)$ to hold. ■

4.5 Putting it all together

We established in the previous sub-section that for $T \in \mathcal{T}$ and $A \in \mathcal{S}_{seq}$, $safe(A, T)$ is true if $live\_in(T)$ is complete for $length(T)$ instructions and $live\_in(T) \subseteq A$. That is, a task is safe if its live-ins are complete and consistent with architected state. In this sub-section, we refine our formal model of MSSP execution to replace predicate $safe(A, T)$ with checks for completeness and containment.

The check for completeness is easily performed by the slave processor: if it cannot find (in its local read/write store) the machine state required for the execution of an instruction, completeness does not hold and the slave aborts its task. Thus, if a slave successfully reaches the end of its task, we can be sure its live-in set is complete. Containment is established by the verify unit when it receives the live-in sets from slaves; it can check each member of the live-in set against the corresponding member of architected state. We therefore need to make changes to Definitions 3.6 and 3.9; Definition 3.5 remains as is.

Definition 4.8 (Slave execution, refined) *For $T \in \mathcal{T}$ such that $state(T) = \texttt{RUN}$, we define:*

$$slave\_step(\langle S_{in}, n, S_{out}, k, \texttt{RUN}\rangle) = \left\{ \begin{array}{ll} \langle S_{in}, n, \emptyset, 0, \texttt{ABORT}\rangle & \text{if } S_{out} \text{ is not complete} \\ \langle S_{in}, n, seq\_step(S_{out}), k+1, \texttt{RUN}\rangle & \text{if } k < n \\ \langle S_{in}, n, S_{out}, k, \texttt{COMMIT}\rangle & \text{otherwise} \end{array} \right]$$

The function is undefined for all tasks T with $state(T) \neq \texttt{RUN}$. □

Recall, the predicate $done(T)$ was defined to be true if and only if $state(T) \neq \texttt{RUN}$, so it becomes true when a task's state becomes either `COMMIT` or `ABORT`. The *slave_step* function therefore guarantees progress toward completion of the task, either by moving it to state `COMMIT` (if completeness checks all pass), or state `ABORT` (if completeness chacks fail). At that point, the machine will attempt to commit the task.

Definition 4.9 (Commit, refined) *For $A \in \mathcal{S}_{seq}$ and $T \in \mathcal{T}$, where $done(T)$ is true, we define:*

$$commit(A, T) = \left\{ \begin{array}{ll} A \leftarrow live\_out(T) & \text{if } live\_in(T) \subseteq A \text{ and } state(T) = \texttt{COMMIT} \\ A & \text{otherwise} \end{array} \right]$$

□

5 Memory-like state

Our formalizations have, up to this point, implicitly relied upon a number of properties holding for the machine state that is manipulated during instruction execution. More specifically, we have been assuming all along that the state manipulated by execution comprises general purpose registers and main memory only. In real machines, this is, of course, only part of the picture; our results in the preceding sections therefore apply only to a subset of machine state that might be encountered by a real application.

In this section, we enumerate the various requirements that have so far been implicit in our work. That is, we make explicit the preconditions for our results in Sections 3 and 4. These preconditions relate primarily to machine state, and it is the extant definition of MSSP execution that depends on their holding true. Accordingly, we extend our definition of MSSP execution to deal with situations in which those preconditions do not hold. This, in turn, requires a slight refinement of our SEQ model.

5.1 Memory-like state

To understand the potential for the problems alluded to above, consider the verify/commit phase of MSSP execution. We make at least the following assumptions in our formal model:

- At verification, we assume checking $live\_in(T) \subseteq A$ can be performed without changing A; and

- Definition 4.4 does not specify the order in which the superimposition operator replaces members in its left operand with content from its right operand, and thus assumes all orderings are equivalent; and
- Lemma 4.4 presumes that writes to the same storage cell can be coalesced into a single write to produce the same net effect as their serialized application.

Although these assumptions are reasonable for the most common forms of machine state, we must acknowledge the existence of state that does not have these properties. For example, reads from and writes to memory-mapped I/O addresses usually have side-effects, either within the machine state itself, or external to the machine. We require, therefore, a more complete model of machine state that distinguishes two types of storage cells.

Definition 5.1 (Machine state) *We define* $\mathcal{S}_{seq} = \mathcal{S}_{mem} \cup \mathcal{S}_{io}$*, where*

$$\begin{array}{lcl} \mathcal{S}_{mem} & = & \{S \subseteq \mathcal{N}_{seq} \times \mathcal{V}_{seq} : \mathit{well\_formed}(S) \wedge \mathit{memory\_like}(S)\} \\ \mathcal{S}_{io} & = & \mathcal{S}_{seq} - \mathcal{S}_{mem} \end{array}$$

To distinguish elements from these two sets, we define disjoint namespaces for each:

$$\mathcal{N}_{mem} = \bigcup_{S \in \mathcal{S}_{mem}} \mathit{names}(S)$$

$$\mathcal{N}_{io} = \bigcup_{S \in \mathcal{S}_{io}} \mathit{names}(S)$$

□

Thus, we have partitioned $\mathcal{S}_{seq}$ into two disjoint components: $\mathcal{S}_{mem}$, which is the set of all machine states that are *memory-like* [1], and $\mathcal{S}_{io}$, which contains all remaining members of $\mathcal{S}_{seq}$ (that is, machine states that are not memory-like). Note that because $\mathcal{S}_{mem} \cup \mathcal{S}_{io} = \mathcal{S}_{seq}$, it follows that $\mathcal{N}_{mem} \cup \mathcal{N}_{io} = \mathcal{N}_{seq}$.

Definition 5.2 (Memory-like state) *For* $S \in \mathcal{S}_{seq}$*, we define the predicate* $\mathit{memory\_like}(S)$ *to be true if and only if all of the following hold.*

1. *All members of* S *can be both read and written;*
2. *Neither reads nor writes have any side-effects;*
3. *A write followed by a read yields the written value.*

□

Our work in Sections 3 and 4 implicitly assumed these properties for all members of $\mathcal{S}_{seq}$. With the above definitions, we have merely made it explicit that only a subset of $\mathcal{S}_{seq}$—namely, $\mathcal{S}_{mem}$—behaves as required. With $\mathcal{S}_{seq}$ thus partitioned, we are now ready to revisit our SEQ and MSSP models to identify precisely the domains in which execution in each is defined.

5.2 SEQ execution

Recall, SEQ serves as the reference against which we measure correctness of MSSP. Accordingly, if our results are to be meaningful, we require that SEQ be representative of a realistic ISA. It must, therefore, support execution of instructions that might read or write state that is not memory-like. Since execution in SEQ modifies machine state one instruction at a time, and in the order specified by the program, none of our formalisms are predicated on memory-like behaviour being adhered to. That is, any non-memory-like behaviour resulting from reads from, or writes to, members of $\mathcal{S}_{io}$ are, by definition, intended to occur in SEQ; side-effects and ordering constraints form part of the machine's sequential ISA specification.

We therefore maintain our formalization of sequential execution, as per Definition 3.2. However, our subsequent changes to the MSSP model do require that we modify the definition of single-instruction execution (Definition 4.5). We need this change to allow us to reason about the subset of machine state that participates in the execution of an instruction.

Recall, $\mathit{seq\_step}(S) = S \leftarrow \delta(S)$. We now interpret δ according to the fetch-decode-execute model described earlier.

Definition 5.3 (Instruction execution, refined) *We define $\delta : \mathcal{S}_{seq} \mapsto \mathcal{S}_{seq}$ as follows.*

$$\delta(S) = execute(fetch(S))$$

The functions $fetch : \mathcal{S}_{seq} \mapsto \mathcal{S}_{seq}$ and $execute : \mathcal{S}_{seq} \mapsto \mathcal{S}_{seq}$ remain uninterpreted. □

Informally, the *fetch* function "reads" from machine state all those name-value pairs that will participate in execution of the next instruction. The resulting set is operated on by function *execute* to produce the results of an instruction's execution. (This is the decode-execute part of the execution model.) These are written back to machine state by the superimposition of $\delta(S)$ on S.

5.3 MSSP execution

MSSP execution can occur only with state that is memory like. We have already alluded to the reasons for this constraint; to be clear, they are as follows.

- Slaves cannot speculatively operate on members of $\mathcal{S}_{io}$, simply because their speculation must be checked by the verifier, which, in turn, must read architected state; it may not be possible to do so without affecting that state (reads might have side-effects).
- Again, because side-effects are possible, it is not in general safe to coalesce multiple writes into a single update if each of those writes would have had a (necessary) side-effect.

We address these issues by circumventing the problem: we will refine our MSSP model to operate only on members of $\mathcal{S}_{mem}$.

We admit two "modes" of execution into the MSSP model. State is modified using the SEQ model when it is detected that the next instruction will "touch" $\mathcal{S}_{io}$; otherwise, MSSP execution occurs as defined previously. Before presenting this refinement of the MSSP model, we formalize the notion of state that is touched by the next instruction.

Definition 5.4 (Instruction footprint) *The set of name-value pairs that participate in the execution of the next instruction is called the footprint of that instruction. Formally, we define $footprint : \mathcal{S}_{seq} \mapsto \mathcal{N}_{seq}$, as follows.*

$$footprint(S) = names(fetch(S)) \cup names(execute(fetch(S)))$$

□

MSSP-mode operation can safely occur when the next instruction's footprint is wholly within the memory-like namespace; that is, when $footprint(S) \subseteq \mathcal{N}_{mem}$, or, equivalently, when $footprint(S) \cap \mathcal{N}_{io} = \emptyset$. It remains to be specified how this requirement is enforced.

Since slaves execute entirely within their local read/write store, which is isolated from architected state, whether or not they touch $\mathcal{S}_{io}$ is not important; as noted earlier, it is during the verify/commit process that problems with $\mathcal{S}_{io}$ arise. We therefore constrain MSSP execution to remain within $\mathcal{S}_{mem}$ by checking the members of the live-in and live-out sets before permitting verification and commit to proceed. If this check finds members of $\mathcal{S}_{io}$, we simply discard the offending task. The proposed check is formalized as follows.

Definition 5.5 (Clean tasks) *We say $T \in \mathcal{T}$ is clean, and write $clean(T)$, if its live-in and live-out sets are both memory-like. Formally:*

$$clean(T) \Leftrightarrow (live\_in(T) \in \mathcal{S}_{mem} \wedge live\_out(T) \in \mathcal{S}_{mem})$$

Equivalently:

$$clean(T) \Leftrightarrow (names(live\_in(T)) \subseteq \mathcal{N}_{mem} \wedge names(live\_out(T)) \subseteq \mathcal{N}_{mem})$$

□

This leads us to a new specification for MSSP operation. Our machine continues to advance architected state, as before, until it reaches a configuration in which the next instruction touches $\mathcal{S}_{io}$. At that point, the machine advances its architected state according to the SEQ model, one instruction at a time. Further, while in MSSP-mode, we filter tasks that have touched state outside of $\mathcal{S}_{mem}$ before passing them onto the verify/commit unit. The following definition captures this idea.

Definition 5.6 (MSSP execution, refined) *The function $mssp : \mathcal{S}_{seq} \mapsto \mathcal{S}_{seq}$ is defined as follows.*

$$mssp(A, [T]||\tau) = \left\{ \begin{array}{ll} seq\_step(A) & \text{if } footprint(A) \cap \mathcal{N}_{io} \neq \emptyset \\ mssp(A, \tau) & \text{if } clean(T) \text{ is false} \\ mssp(commit(A, T), \tau) & \text{if } done(T) \\ mssp(A, mssp\_step([T]||\tau)) & \text{otherwise} \end{array} \right]$$

Functions $commit : \mathcal{S}_{seq} \times \mathcal{T} \mapsto \mathcal{S}_{seq}$ and $mssp\_step : \mathcal{T}^* \mapsto \mathcal{T}^*$ are defined as before. □

Implicit in the above definition is the assumption that the MSSP machine is able to check if $footprint(S) \cap \mathcal{N}_{io}$ is empty and that $clean(T)$ holds without actually reading state that is not memory like. We take it to be a reasonable assumption that a machine is able to do so.

5.4 Equivalence

Equivalence of the modified MSSP and SEQ models still holds because Theorem 3.2 can be applied to any $\tau \in \mathcal{T}^*$ that contain only clean tasks. Execution in all other cases either does not affect architected state, or advances it according to the sequential model, in which case equivalence follows trivially. ■

6 Conclusion

This report presents the formal verification of MSSP, a recent proposal for speculative parallelization of sequential programs. Our primary objective has been to formally establish that MSSP is capable of achieving the equivalent of a sequential execution. We demonstrated this in Section 3, where we showed that any state reachable by an MSSP machine is also attainable by a sequential machine implementing the same ISA as the MSSP slaves. That result was predicated on the notion of task safety, a property we initially assumed could be checked directly by the MSSP machine. We then proved in Section 4 that task safety follows from a more low-level set of checks: completeness and containment of live-ins within architected state, which we argued can be performed by the slave processors and the verify/commit unit, respectively.

In establishing these results, we also introduced a model for a hypothetical MSSP machine (Section 3.2). We believe this model will serve a useful purpose in subsequent developments in the evolving MSSP paradigm. Specifically, a number of performance-enhancing modifications to the existing MSSP definition are imminent; our hypothetical machine, being sufficiently agnostic to performance concerns, will not likely incur similar changes. As a result, it can serve as a reference against which design changes can be checked in terms of correctness.

In addition to analyzing the correctness of MSSP, we identified conditions in which it cannot work. The latter issue was addressed in Section 5, where we enumerated the requirements we impose on machine state to ensure that it is amenable to MSSP-style execution. In particular, we identified memory-like behaviour as the key property upon which we depend. While this does imply MSSP is not universally applicable, it also admits the possibility for MSSP operation crossing the user-kernel boundary. Simply put, execution of operating system code is no different from user code in our formal models: in both cases, instructions transform machine state as per the machine's ISA. That MSSP can execute kernel code—so long as that code touches only memory-like state—is a boon to the paradigm, since it potentially allows slaves to take interrupts, handle exceptions and make system calls. This is an area we plan to investigate thoroughly in the near future.

Additional challenges are posed by certain machine components that we would prefer not to incorporate into our formal models. The TLB is one such example. By including the TLB in a machine's state, we would be architecting its content. Doing so would force MSSP tasks to perform the same series of TLB replacements as a sequential execution, which, in turn, would require that the master also predict TLB replacements. Clearly, such an architecture would suffer unnecessary performance degradation through spurious misspeculations. The problem arises because the exact mix of entries in the TLB is not important from a correctness point of view; what counts

is that the mappings that are used are consistent with the page table held in architected state. One of our current research objectives, therefore, is to develop an abstract definition of the TLB that captures, instead of its exact state, the semantics of its role in supporting virtual memory.

We have also drawn attention in this report to the fact that MSSP is an example of an architecture that decouples performance and correctness by dedicating distinct hardware to each concern. In this context, our results make the important contribution that such a decoupled machine is indeed feasible, at least in terms of isolating correctness from performance concerns. Specifically, by modelling the master processor as a random generator of live-in data, we successfully show that MSSP's correctness is independent of fast-path components.

References

[1] R.L. Sites, editor. *Alpha architecture reference manual*. Digital Press, 3rd edition, 1998.

[2] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *Proc. 22nd Annual International Symposium on Computer Architecture*, June 1995.

[3] C. Zilles. *Master/slave speculative parallelization and approximate code*. PhD thesis, University of Winsconsin - Madison, 2002.

[4] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *Proc. 35th Annual ACM/IEEE International Symposium on Microarchitecture*, November 2002.